

MDS: The
Strategic
Advantage
for Future
Flight
Projects

MDS

Mission Data System

```
#include "TransportableGoal.h"
#include "SimpleProduct.h"
namespace Mds {
namespace Mds {
namespace Fw {
namespace Dm {
namespace Examples {
using namespace std;
/**
 * Need to statically define my
 */
const ProductClassID
SimpleProduct::myClassID(999);
SimpleProduct::SimplePro
string& name_arg,
int source_arg,
const string& stuff)
u may need...
```



```
#include "TransportableGoal.h"  
#include "SimpleProduct.h"
```

```
namespace Mds {
```

```
    namespace Fw {  
        namespace Dm {  
            namespace Examples {
```

```
                using  
                namespace std;
```

```
            /**  
             * Need to statically define r  
             * ProductClassID.  
             */
```

```
                const ProductClassID  
                SimpleProduct::myClassID(999)
```

```
                SimpleProduct::SimpleProduct(  
                string& name_arg,  
                int source_arg,  
                const string& stuff)
```

A New Approach 2

8 Design Philosophy

MDS Benefits 16

20 MDS Implementation

Research & Commercial
24 Opportunities



C o n t e n t s

We are entering a new era of solar system exploration. Until recently, deep space missions tended to be one-of-a-kind, with distinct science objectives, instruments, and mission plans. Missions were spaced years apart, and mission software was developed independently for communications, commanding, attitude control, navigation, and other recurring tasks.

Software design also was limited by the radiation-hardened flight processors that were years behind their commercial counterparts in speed and memory, limiting performance and science returns.

The need for a new approach to software development became apparent with the demand for smaller, low-cost deep space missions. When the Jet Propulsion Laboratory (JPL) launched six missions in six months between October 1998 and March 1999, there was no common framework for developing mission software and little software reuse; each mission either built the software from scratch or tried to reuse software that was never designed for reusability.

MDS
architectural
design creates
an approach
in which soft-
ware engineers
and systems
engineers
communicate
in a common
language.

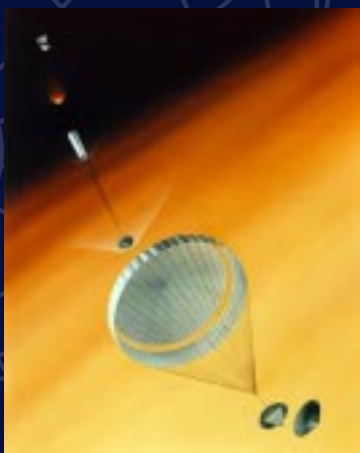
A New

Approach



Future missions require a new vision

MDS may be the single answer for developing software for future mission requirements. Using our current practices, missions will remain costly if software cannot be reused efficiently. Science returns will be limited if resources are underutilized. And missions will require more autonomous capabilities for in situ exploration. MDS addresses these issues and offers a solution by unifying software and systems engineering. The MDS approach makes reuse possible, provides resource management to maximize spacecraft returns, and supports autonomous capabilities.



Future missions obviously will require reusable software. This software must accommodate more complex, autonomous science returns and easy infusion of new technologies. Our challenge is to design reliable software that expands mission capabilities and that can be reused efficiently and effectively in various mission scenarios.

In April 1998, JPL began to address this challenge. The Laboratory initiated the Mission Data System (MDS) to rethink the mission software life cycle and to develop software architectures that accommodate the complexities of future mission requirements. Three years of thoughtful effort have produced a unified flight, ground, and test data system architecture that is revolutionary in scope and vision. This component-based, object-oriented design assimilates generations of JPL's domain knowledge and addresses several mission needs: It allows software reuse, expands autonomous capabilities for in situ exploration, and establishes a basis for infusion of new software technologies.

Meeting New Mission Demands

As we launch more missions in shorter timeframes, we need software that can be reused from mission to mission. MDS can accomplish this by taking advantage of the

recent advancements in flight computers. These computers allow flight and ground elements to share the same architecture and accommodate a common framework — a key component of the MDS software approach. This common framework eliminates duplication of software development, simplifies the integration of new elements, and allows many ground capabilities to be migrated to flight.

Missions are evolving from reconnaissance and mapping at safe distances to in situ exploration in dynamic and unpredictable environments. To produce successful science returns, onboard resources must be managed and data must be assessed in real time to make on-the-spot changes for further observations. These autonomous capabilities require large investments of time and money — investments that individual projects cannot reasonably sustain.

The MDS architecture significantly improves software design. This object-oriented design can integrate new capabilities easily; it also gives software engineers the tools to express mission and systems engineering concepts in a natural language, creating software that is flexible, easily maintainable, and, most important, reliable.

MDS promises to meet new mission demands by expanding mission capabilities, increasing scientific returns,

and providing a foundation for reusable software. The philosophy, application, and various benefits of MDS are described in the following sections.

Complementary Engineering Approach

As technology changes, so must our engineering practices. Traditionally we have separated systems engineering from software development. However, software engineering and systems engineering are highly interdependent. Systems engineering must understand what the system is supposed to do, whereas software engineering must understand how the system will do it. In other words, software manages a system's capabilities and resources, whereas systems engineering identifies the required capabilities and resources.

The MDS architecture supports this complementary approach. The design allows software engineers and systems engineers to communicate within a common “language,” in which software abstraction is defined within the language of systems engineering. As a result, software engineers and systems engineers share a common approach to defining, describing, developing, understanding, testing, operating, and visualizing

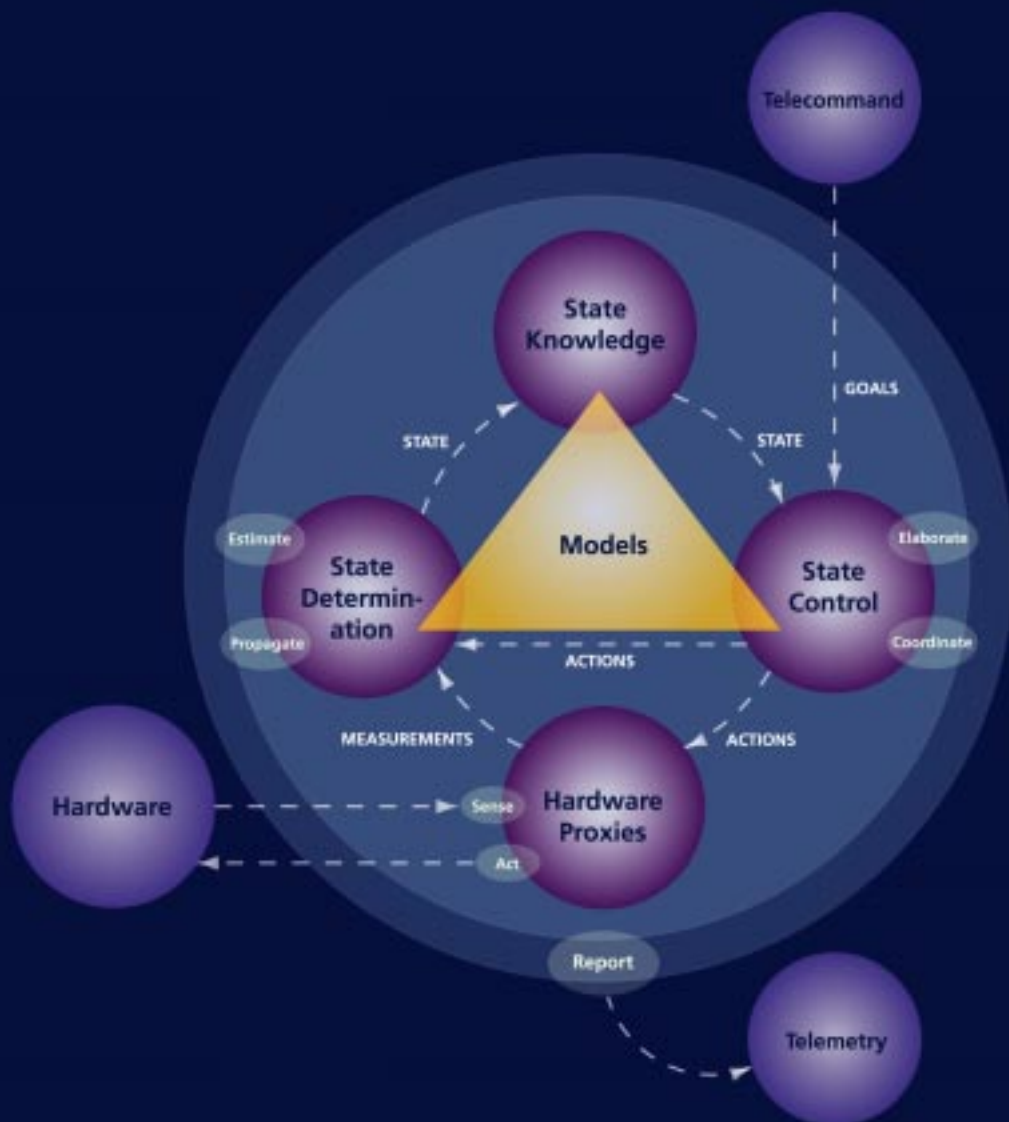
what systems do. The net result: systems that are more reliable, cost-effective, and reusable.

MDS Core Product

The MDS core product is a unified architectural framework for building end-to-end flight and ground software systems. This framework includes the necessary elements for building goal-oriented, autonomous commanding; intelligent data management and transport; integrated guidance, navigation, and control, and most other capabilities needed for mission software. MDS core products include design patterns for adapting the framework for

software mission functions. Customers also receive a set of pre-integrated and pretested frameworks, complete with executable example uses of those frameworks running a simulated mission. The design is object-oriented, and the framework design is expressed in Unified Modeling Language (UML).

Customers who use the MDS framework can focus on mission-specific design and development without having to create and test a supporting infrastructure. As MDS matures, new and innovative design concepts will be incorporated in the framework — increasing the framework's capabilities.



The MDS architecture is designed for goal-oriented control, enabling autonomous robots — spacecraft, rovers, and ground stations. Externally, MDS takes goals as input and interacts with hardware to achieve the goals. Internally, the architecture emphasizes the central role of state knowledge and related models. Together they serve estimators and controllers in a disciplined structure for closed-loop control. Telemetry provides external visibility of internal activities.

The MDS framework evolves from a new way of thinking about how software engineering is practiced. The design philosophy MDS has adopted is based on the best practices from various disciplines like control systems, robotics, data networking, software engineering, and artificial intelligence. This design unifies systems engineering and software engineering practices — a desperately needed capability. The design philosophy is based on 13 key themes, each expressing an innovative design approach to solving specific problems.

Theme 1: Construct subsystems from architectural elements, not the other way round.

Problem MDS Addresses: A priori partitioning of software development tasks along traditional subsystem boundaries encourages multiple, point-design solutions, decreasing efficiency and reuse value and complicating integration.

The traditional approach to software design for deep space missions has been to compartmentalize the work, resulting in individual software engineering teams applying their own solutions to common problems. The specialized products that result have minimal reuse value and require many iterations to integrate with other subsystems. MDS reverses this process by identifying common problems and then posing common solutions that can be tailored to each particular application. This collection of common solutions is referred to as the

State is
the unifying
concept
for MDS.
Modeling the
system and
its state is
important,
and the MDS
approach does
this very
effectively.

```
namespace Examples {  
  
    using  
    namespace std;  
  
    /**  
     * Need to statically define my  
     * ProductClassID.  
     */  
    #include "Transportable.h"  
    #include "SimpleProduct.h"  
  
    const ProductClassID  
    SimpleProduct::myClassID(999);  
  
    namespace FW {  
        namespace DM {  
            SimpleProduct::SimpleProduct(const  
            strings name_arg,  
            int source_arg,  
            using  
            const strings& stuff  
            namespace std;  
            *  
            Product(name_arg), source_id(source_arg)  
            /** any other  
            initialization you may need... */  
            *need to static  
            ProductClassID.  
            *  
            *  
            const ProductClass  
            SimpleProduct::myClass  
            SimpleProduct::~SimpleProduct() {  
                *  
            }  
  
            SimpleProduct::Si  
            strings& name_arg,  
            SimpleProduct::SimpleProduct(Ser::Object  
                : Product(in,  
                myClassID), source_id(in.readI32()), dat  
                *  
            Product(name_arg), se  
            /** all you
```

Design

Philosophy

MDS framework. A fundamental driver in selecting this framework is the recognition that space system designs are always tightly coupled. Constrained resources demand it, so managing these interactions is the foundation of good software design in such systems. The MDS frameworks have been designed from the outset to uniformly address this need.

Theme 2: Migrate capability from ground to flight, when appropriate, to simplify operations.

Problem MDS Addresses: When flight and ground software systems are designed and implemented as distinct disciplines, operational constraints of deployment across platforms are not addressed, preventing easy migration between ground and flight. However, this migration is essential to flight system autonomy.

Increasingly powerful flight processors make it possible to migrate functions that have traditionally been performed on the ground to a spacecraft or rover. This migration might occur after launch — after ground operators have gained experience with the vehicle and have decided that some activities can be automated without further human-in-the-loop control. In some cases, the same code may be used for both flight and ground control. However, even in cases where flight implementations are different because they exploit the immediacy of their interactions with the spacecraft, the uniformity of

addressing other system elements permits these migrations to take place with minimal perturbation to the rest of the system.

More importantly, there is a need for such migration to accomplish missions that must react quickly to events without earth-in-the-loop, as is the case with an autonomous landing on a comet or rover explorations on the surface of Mars. By adopting a unified architecture, we can address a wide range of mission possibilities using a single MDS framework. To accomplish this, both flight and ground capabilities must be designed within a shared architecture.

Theme 3: System state and models form the foundation for information processing.

Problem MDS Addresses: To provide an architecture that is capable of spanning a wide variety of domains and to organize a systems engineering and design process that is structured and conducive to software adaptation, we need a unifying theme for describing expected and desired system behaviors.

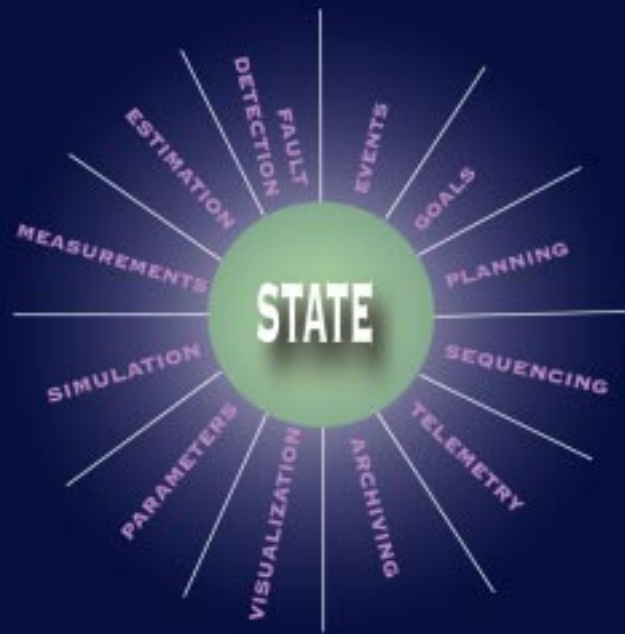
MDS is a state-based architecture — state and models are central to the MDS framework. State describes the momentary condition of an evolving system, and models describe how those states evolve in time. Together, state and models supply all the information needed to operate a system, predict future state, control toward a desired objective, assess performance, and more.

In a typical MDS adaptation, the totality of state representations provides a representation of the total system that is sufficiently complete to provide adequate knowledge of state for all purposes. While there may be elements of a project outside the MDS purview, even the external elements are described at least by their behavior. In all cases, state is accessible globally in a uniform way through state variables, as opposed to a program's local variables.

Theme 4: Express domain knowledge explicitly in models rather than implicitly in program logic.

Problem MDS Addresses: Domain knowledge about mission instruments, actuators, sensors, plumbing, wiring, and other elements must be consistently expressed from mission to mission if flight software is to be reused. Conventional practice develops programs in which the logic implicitly contains this domain knowledge but expresses the knowledge in a “hidden” form that is difficult to validate and reuse.

MDS expresses domain knowledge more explicitly in inspectable models. These models can be any of several forms, as long as they separate the domain knowledge from the general logic for applying that knowledge to solve a problem. The task of customizing MDS for a mission, then, becomes largely a task of defining and validating models.



System state is the architectural centerpiece for information process-

ing in MDS. This state-based architecture allows complex system behavior to be captured in a simple, straightforward design.

Theme 5: Operate missions using specifications of desired state, rather than as sequences of actions.

Problem MDS Addresses: Mission operations procedures that are based on detailed specification of sequences of actions limit onboard autonomy, underutilize onboard resources, and increase operations cost.

Traditionally, spacecraft are controlled through linear (nonbranching) command sequences that are carefully designed on the ground. This approach, however, is difficult for two reasons. First, ground personnel must accurately predict spacecraft state for the time at which the sequence is scheduled to start and throughout its execution. Second, in the event that the actual spacecraft state departs from the predicted state, the sequence must be designed to terminate early and trigger a “safing” response rather than risk that the continued sequence execution will result in harm.

Even when some branching is allowed, it has generally been highly constrained because developing such a sequence amounts to writing a program — not the best approach to routine operation. For this reason, even such limited measures have generally been restricted to highly critical activities.

In contrast, MDS controls both flight and ground state through goals. A goal is expressed as a constraint on a state over some time interval. Unlike a command, which says what to do, a

goal specifies an intent in the form of desired or acceptable states. Similarly, timing is controlled by constraints on when goals apply. Unlike a timed command, where the time to issue the command is specified precisely, goals may apply over flexible time intervals, depending on events or on other system activities, as adjusted automatically by the system.

Goal-directed operation is simpler than traditional sequencing because a goal is easier to specify than the actions needed to accomplish it. More importantly, goals specify only success criteria; they leave options open about the means and timing of accomplishing the goal and the possible use of alternate actions to recover from problems.

Theme 6: Design for real-time reaction to changes in state rather than for open-loop commands or earth-in-the-loop control.

Problem MDS Addresses: An open-loop command sequence used for high-level operations leaves the system vulnerable to unanticipated changes due to variable system behavior, faults, or an uncertain environment. For effective in situ exploration, this practice is undesirable, and even in interplanetary space leads to unsatisfactory performance.

Goal-directed operation implies closed-loop control. This means that the steps taken to achieve goal success may be reconsidered and ad-

justed by the system in response to an immediately observed state of the system. In MDS parlance, a state controller is called a goal-achieving module (GAM). A GAM controls state by comparing present state to desired state, deciding how to change the state, if necessary, and then issuing either sub-goals to lower-level GAMs or issuing direct low-level actions (i.e., primitive actions). These steps may be taken well in advance of the initiating goal, if necessary. When a GAM accepts a goal, it must either achieve the goal or responsibly report that it cannot. A GAM's logic can be arbitrarily simple or sophisticated, but it must always keep the goal issuer informed about the goal's status.


Most GAMs achieve their goals by issuing sub-goals, thus creating a hierarchy of GAMs that terminates in primitive actions. GAMs can report why they acted as they did in terms of what discrepancies between state and goals prompted action, and what sub-goals or commands were issued in response. Since GAMs are self-checking by definition, goal failures will be overtly visible (through goal status) during testing.

Theme 7: Fault protection must be an integral part of the design, not an add-on.

Problem MDS Addresses: Fault protection has generally been viewed and designed separately from the nominal system control functions. Moreover, the fault protection design usually lags behind the nominal system, which is designed with little consideration to its needs. As a result, fault protection generally gets little help — and often only receives interference — from the underlying system.

Fault protection by its nature is a closed-loop process that responds to observed conditions of a system. This process is fundamentally incompatible with an open-loop model of command sequencing, although it is intrinsic to a goal-directed operation. In MDS, fault protection can be woven seamlessly into the larger fabric of robust control.

Goal-achieving modules in MDS need some minimum level of fault detection to enable them to report when an active goal is not being achieved due to a fault. Fault detection is provided through state determination in which fault monitoring is an integral part. Because fault protection arrives concurrently for testing as part of the nominal system functionality, it can be an extremely valuable aid to system testing.



Fault protection is typically an add-on. One mission team spent six months in extensive testing before enabling fault protection. The result was surprising: numerous design errors were detected in the control system. The mission team learned more in a single month running fault protection than they had in six months of intense testing.

Theme 8: Resource usage must be authorized and monitored by a resource management mechanism.

Problem MDS Addresses: Currently our spacecraft capabilities are vastly underutilized because of concerns about maintaining resource margins. Onboard resource monitoring and management is limited, yet is essential to avoid over-subscription of limited resources.

Liberal use of resources without safeguards is potentially hazardous. Consequently, ground operators are cautious about overextending resources such as power and propellant, and tend to maintain large margins of these resources.

Underutilizing resources limits the amount of science data acquisition and return, especially during time-constrained activities like a fly-by or a science experiment with short-lived instruments. Resource management is also a major concern during in situ exploration or fault responses when unanticipated factors can affect resource usage.

MDS addresses these problems through a resource management mechanism that avoids overuse. Limited resources are prioritized by goals, so if there is no other recourse — such as in rescheduling — the system can arbitrate among contenders how resources should be used. Because resource managers always know how much of each re-

source is subscribed, resource usage can be maximized, thereby maximizing performance, while remaining safe.

Theme 9: For consistency, simplicity and clarity, separate state determination logic from control logic.

Problem MDS Addresses: A typical but ineffective practice in software development is to co-mingle control logic with state-determination logic. Assessments of the current state, upon which decisions are made, are often implicit and difficult to inspect and verify. This co-mingling of logic also confounds attempts at reuse.

Architecturally, MDS separates state determination from state control, which are coupled only through state variables. State determination is the process of interpreting measurements and other data to generate state knowledge supplied to a state variable as an estimate. Control, in contrast, attempts to achieve goals by issuing commands and sub-goals that should drive estimated state toward desired state. Keeping these two tasks separate simplifies design, programming, and testing, and allows for reuse and independent improvements.

Theme 10: State determination must be honest about the evidence; state estimates are not facts.

Problem MDS Addresses: Conflicting or incomplete information is an unavoidable source of trouble that is dangerous if ignored.

State values are rarely known with certainty due to elements like conflicting evidence, characteristic degradation of sensors, failures, and periods of rapid dynamic change. Disastrous errors can result when control decisions are based on highly uncertain state values. For example, it is unwise to perform a main-engine burn when uncertainty in the estimated position of the engine gimbals is large. Problems also can arise from incompatible decisions if different parts of the system react to different evidence.

In the MDS approach, all knowledge of any particular state is captured in a single-state variable. Moreover, a level of certainty accompanies every state estimate so that control can take into account the certainty level when using state values. If certainty drops below some context-specific minimum, then control must react appropriately, perhaps by acquiring additional data or attempting an alternate approach or by abandoning a goal entirely.

MDS Benefits

Using the high-performance, commercially standard flight processors now available, MDS can eliminate redundancy in the development of flight, ground, and test software, reducing mission costs, increasing reliability, and expanding mission capabilities.

The MDS architecture allows missions to construct, from a common set of core elements, a wide variety of software for complex applications — entry-descent-landing (EDL), formation flying, in situ exploration, cross-link vehicle-to-vehicle operation, and more. Mission designs have many things in common, so the design process can be complementary from one mission to the next and the same structures apply to all. Because engineers apply the same fundamental mechanisms (state estimation, goal-based operation, and closed-loop control) to solve problems, they can use this common framework to instantiate these solutions. Details may vary; however, much of the basic structure can be inherited. Common patterns will guide more rapid development, and specific designs may be reusable because the structure in subsequent systems will be similar.

The MDS approach is complemented by an innovative design philosophy that expands software capabilities and will solve many of our future software mission issues.

These are examples of the benefits of MDS design —

Goal-Based Commanding Simplifies Operations

MDS architecture defines and implements goal-based operations to allow mission operators to make direct state-ments of intent (as constraints on state) when commanding the system. This technique:

- Provides a clean separation between “what” and “how.”
- Allows simple to sophisticated onboard fault handling.
- Enables autonomous recovery of activities, even in noncritical situations.

Onboard Resource Management Maximizes Spacecraft Capabilities

Resources can be managed on board and in real time; thus:

- Resources can be deliberately but safely oversubscribed.
- Prioritized goals allow more important activities to run up to “real” rather than “predicted” resource limits.

Reusable Core Software Improves Multimission Design and Implementation

MDS builds on a common framework; the same core software can be used for many missions.

- MDS capitalizes on lessons learned and automatically passes them on to future missions by incorporating them into the multimission software.

- As software is used and reused for subsequent missions, the software evolves and improves.

MDS Design Promotes Better Practices

MDS encourages better engineering practices through a software framework that:

- Requires engineers to think through estimation and control as separate tasks and facilitates their interaction.
- Defines several design patterns to facilitate clean, safe designs.

State-Based Systems Engineering Creates a Common Approach

State-based systems engineering enables systems engineers and software engineers to share a common approach in defining, developing, testing, operating, and visualizing what systems do.

- The MDS framework requires that state, measurements, constraints, telemetry, plans, etc., be represented as software objects.

- In this framework, the language of systems engineering becomes the language of design for adaptations of MDS reusable software.

Faster Technology Infusion Supports Autonomous Capabilities

MDS uses a uniform architecture for flight and ground systems.

- Ground capabilities can be migrated without rewriting them.
- Migration can take place in the same mission, reducing infusion time from years to months.

Theme 11: Separate data management duties and structures from those of data transport.

Problem MDS Addresses: Data management and data transport share a tightly intertwined heritage in space systems as a result of the limited resources available to these functions in earlier systems. The coupling of these two capabilities limits their evolution and flexibility and complicates integration.

MDS distinguishes clearly between data management and data transport. The former elevates data products as entities in their own right, rather than as units of transport. In MDS, data products are software objects that can be created, updated, compressed, summarized or aged, whether or not they are destined for transport. In fact, data management is a service that transcends the flight-ground divide so that data products are treated consistently in both places. Data transport, in contrast, can access any data product — regardless of its internal structure — and serialize it for transport between flight and ground. Packet formats and link protocols are completely hidden from the level of data management. Decoupling these two capabilities keeps the design and testing simpler for each and allows for independent improvements.

Theme 12: Navigation and attitude control must build from a common mathematical base.

Problem MDS Addresses: Navigation and attitude control, though required to solve many similar problems, have traditionally not worked as tightly as they might.

In interplanetary space, navigation and attitude control operate on vastly different time scales and their dynamics don't greatly affect each other. In upcoming deep space missions, the required coupling will be much tighter. For example, escape velocity near an asteroid is so small that firing thrusters for attitude control can greatly affect trajectory. Likewise, landing on a planetary surface or docking with another vehicle, as in a sample-return mission, requires navigation and attitude corrections on similar time scales.

The same forces influence navigation and attitude control. They are both involved in tasks such as pointing. Both are concerned with dynamics, solving geometry problems, and so forth. As part of the MDS approach in designing common architectural mechanisms for common problems, building a shared set of frameworks for navigation and attitude control is essential. This shared set of frameworks ultimately enables the sort of tightly coupled, six-degrees-of-freedom control required for ambitious future missions.

Theme 13: Design interfaces to accommodate foreseeable advances in technology.

Problem MDS Addresses: Software technology has generally lacked an infrastructure to support its demonstration or evaluation in a fully integrated flight system.

MDS will serve missions for many years to come. As software technology advances are made in control systems, fault detection and diagnosis, planning and scheduling, databases, and communication protocols, MDS must be prepared to exploit these technologies.

Nevertheless, MDS needs to maintain some architectural stability to amortize its cost over its missions. The strategy for achieving this is based on careful design of architectural interfaces, behind which a variety of technical approaches can be used. Specifically, MDS designers have consulted with researchers to understand how software interfaces may need to evolve, and then implemented a restricted subset of the interfaces using current mission-ready technology. When more advanced technology becomes mission-ready, the fuller interface will be implemented in an upward compatible manner, namely, in a manner that still works for interface clients using the restrictive subset. Thus, interface client software is not forced to change on the same schedule as interface provider software.



Missions can use a common set of core elements to construct a variety of software for complex applications — entry-descent-landing, formation flying, in situ exploration, cross-link vehicle-to-vehicle operation, and more.

In the past three years, the MDS project completed the design of the core architecture that will be used for future deep space flight and ground software. Now the project is organizing its development activities — a series of demonstrations to test the architecture’s mission application. Two upcoming demonstrations — Mars entry-descent-landing (EDL) and surface rover in situ operations — will test the MDS framework to ensure its readiness for actual missions. These demonstrations will involve mobility, science instrument control, and experimentation.

The design phase for the EDL demonstration is underway; the details of the rover demonstration are being developed.

Demonstrations

of the MDS

framework in

real mission

scenarios will

help the MDS

team evaluate

the software

development

methodology.

MDS

Implementation

```
...ce Mds {
#include "transportable..."
#include "SimpleProduct.h"
namespace Fw {
namespace Mds {
namespace Examples {
namespace Dm {
namespace Examples {
...ng
using
...ce std;
namespace std;

/**
 * Need to statically define my
...need to statically define my
...class ID.
const ProductClassID
...Product::myClassID(999);

...st ProductClassID
...SimpleProduct::myClassID(999);
...reducer::myClassID(999);
int source;
const string stuff;

1
Product(name_arg), source(source_arg), data(stuff) {
...leProduct::SimpleProduct(const
...name_arg,
int source;
const string stuff)
...SimpleProduct::Simple
}
(name_arg), so...
any other
...SimpleProduct::Simple
...ization... may need
ClassID, source_id(is...
...t::SimpleProduct(const
...int source_arg, dar
...const string st
...id(source...
```


The EDL Demonstration

Entry-descent-landing is a time-critical event that requires complex data processing and decision making. EDL offers no opportunity for ground intervention and requires very rapid fault recovery and high performance. The demonstration is designed to stress the MDS design to ensure that the framework is suitable for this kind of mission application.

Phase 1 of the EDL demonstration will take place in a simulated environment and will cover several scenarios, beginning a few hours before atmospheric entry and proceeding through initial post-landing operations. This phase also will include fault testing.

In phase 2, MDS will work with the Mars team to adapt MDS components to the EDL demonstration. A parallel hardware adaptation will also begin. These efforts will introduce resource management capabilities and thoroughly explore fault-protection issues.

The Rover Demonstration

The second demonstration for the MDS framework is Mars rover in situ operation. Because ground involvement in this operation is intermittent, too slow, or not possible, autonomy is needed to accommodate uncertainty and improve science return. The harsh and variable environment that the rover will traverse results in tight resource margins that will be hard to predict. To test the MDS framework,

the rover demonstration is tentatively planned to address a variety of scenarios like these:

1. Maneuver in a variety of terrains.
2. Plan a series of competing activities for mobility, science, and telecommunication in the presence of constraints like the day-night cycle, telecommunications opportunities, power management, temperature management, and more.
3. Perform various tests that stress the system using faults, margin problems, maneuvering difficulties, etc.
4. Incorporate rover ground systems to operate the rover and its payload.
5. Operate an articulated rover arm.
6. Gather, store, process, and transport simulated science data.

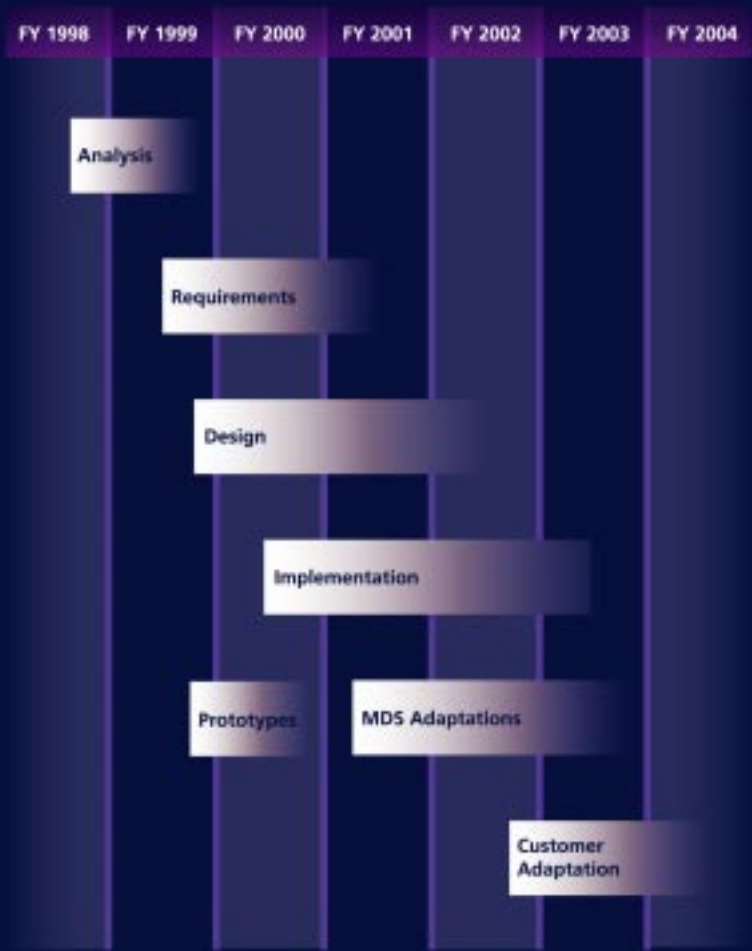
The EDL and rover demonstrations will test the MDS framework in real mission scenarios and will also help the MDS team evaluate the software development methodology, which focuses on using common software, structures, and design process.

Software Development Methodology

MDS uses an iterative life cycle approach in the development and release of all its products. This itera-

tive approach allows the team to divide development efforts into smaller, simpler, and more manageable tasks. Most important, customers are involved in the early development phases. They can work side by side with the MDS team in the early stages of prototyping to ensure that the evolving product meets mission requirements. This process allows

customers to test early prototype versions against various scenarios and make adjustments as the product evolves. As a result, customers and MDS managers are able to judge project progress in terms of functionality. The benefits: Customers have more choices, and because integration is already done, costs are reduced.



The Mission Data System project was formed in April 1998, with the first year’s effort dedicated to the analysis of state-based architecture. Since then, the focus has been on requirement capture, detailed design, and core software development. The next phase will focus on adaptations that demonstrate the framework is suitable for mission operations.

The MDS approach to software engineering holds potential for widespread application in research and industry. Any industrial system that uses closed-loop control for operations in an unpredictable environment could be developed more efficiently using the MDS framework. These systems make a careful accounting of state in order to operate safely. State is a first-order element in the MDS framework; therefore, the framework allows complex system behavior to be captured in a straightforward design. For example, MDS might be used to control a petrochemical plant that requires the coordination of hundreds of valves, pumps, and other components. In the traditional approach, control is translated into coordinated sequences. As the plant grows, emergent behaviors become difficult to predict, and the system becomes vulnerable to faults. In the MDS approach, the system directly implements control in terms of state — no translation is needed. Consequently, the MDS approach is both easier to understand and more fault tolerant.

Industrial and
research
developers
are assured a
quality tool
evaluated
in the most
rigorous of
environments:
spaceflight
testing.

Research & Commercial Opportunities





LICENSING MDS

The California Institute
of Technology (Caltech)

is currently patenting the
MDS software design.
Caltech/JPL will then be
able to offer licenses to
companies who want
to build on the MDS
framework.



```
namespace Mds {  
    #include "Transporta  
    #include "SimpleProd  
    namespace Fw {  
        namespace Mds {  
            namespace Dm {  
                namespace Examp  
                namespace Dm {  
                    namespace Examp  
using
```


MDS might be used in a wide variety of nonflight and research applications, such as:

- Control systems for automobiles
- Avionics systems for commercial and military flight
- Next-generation home appliances
- Physical plants like heating and air conditioning systems
- Industrial processing

The MDS framework also offers a potential advantage to technology programs. Historically, new technology has not been integrated easily into flight programs or other deployed systems. New technology developed on an MDS platform, however, could be migrated easily from the laboratory to an MDS-based deployed system.

Several factors facilitate technology transfer. First, the same framework is used; therefore, customers can

evaluate how the technology performs in real applications. Second, integration of new technology is simplified. Third, once the technology is integrated, it can be used by a number of third parties because MDS can be widely deployed.

Most important, industrial and research developers are assured a quality tool that has been evaluated in the most rigorous of environments — spaceflight testing. MDS will become a more robust product as it is widely used and tested. And, because MDS is being designed for deep space missions, its design must pass a very rigorous test regimen. This adds considerable value. No internal verification program will be able to provide comparable assurance of a solid system core.



```
#include "TransportableGoal.h"  
#include "SimpleProduct.h"
```

```
namespace Mds {
```


```
    namespace Fw {  
        namespace Dm {  
            namespace Examples {
```

```
                using  
                namespace std;
```

```
                /**  
                 * Need to statically define n  
                ProductClassID.
```

```
                */  
  
                const ProductClassID  
                SimpleProduct::myClassID(999
```

```
                SimpleProduct::SimpleProdu  
                string& name_arg,  
                int source_arg,  
                const string& stuff)
```



Future NASA missions will require more sophisticated software than is possible with the current generation of software architectures.

MDS is a significant step forward in unifying software and systems engineering practices. It incorporates modern software development methodologies in a system design based on 35 years of operational spaceflight experience. This new way of thinking and designing software has produced technology that is a strategic step forward in realizing the next generation of mission software.

A Strategic

Step Forward



National Aeronautics and
Space Administration
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL 400-931 01/01